Московский авиационный институт
(национальный исследовательский университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №2 по курсу «Операционные системы»

Управление процессами в операционной системе «UNIX».

Студент: Кварацхелия А. Ю.
Преподаватель: Миронов Е. С.
Группа: М8О-201Б-21
Дата:
Оценка:
Подпись:

Москва, 2023

Условие

Составить и отладить программу на языке Си, осуществляющую работу с процессами и взаимодействие между ними в операционной системе UNIX/LINUX. В результате работы программа (основной процесс) должен создать для решение задачи один или несколько дочерних процессов. Взаимодействие между процессами осуществляется через системные сигналы/события и/или каналы (*pipe*). Необходимо обрабатывать системные ошибки, которые могут возникнуть в результате работы.

Задание

Родительский процесс создает два дочерних процесса. Первой строкой пользователь в консоль родительского процесса вводит имя файла, которое будет использовано для открытия File с таким именем на запись для child1. Аналогично для второй строки и процесса child2. Родительский и дочерний процесс должны быть представлены разными программами.

Родительский процесс принимает от пользователя строки произвольной длины и пересылает их в *pipe*1 или в *pipe*2 в зависимости от правила фильтрации. Процесс *child*1 и *child*2 производят работу над строками. Процессы пишут результаты своей работы в стандартный вывод.

Правило фильтрации: с вероятностью 80% строки отправляются в pipe1, иначе в pipe2. Дочерние процессы удаляют все гласные из строк.

Код программы

utils.h

```
#ifndef UTILS_H
#define UTILS_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char* ReadString(FILE* stream);
char* ReadStringWithoutVowels(FILE* stream);

#endif
```

parent.h

```
#ifndef PARENT_H
#define PARENT_H
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>
#include <sys/wait.h>
#include <sys/stat.h>
#include <sys/types.h>

void ParentRoutine(char* pathToChild, FILE* input);

#endif
```

0.0.1 utils.c

```
#include <utils.h>

char* ReadString(FILE* stream)
{
    if(feof(stream)) {
        return NULL;
    }

    const int chunkSize = 256;
    char* buffer = (char*)malloc(chunkSize);
    int bufferSize = chunkSize;

    if(buffer == NULL)
    {
        printf("Couldn't allocate buffer");
        exit(EXIT_FAILURE);
    }

    int readChar;
    int idx = 0;

    while ((readChar = getc(stream)) != EOF)
    {
        buffer[idx++] = readChar;

        if (idx == bufferSize)
        {
```

```
            buffer = realloc(buffer, bufferSize + chunkSize);
            bufferSize += chunkSize;
        }

        if (readChar == '\n') {
            break;
        }
    }

    buffer[idx] = '\0';

    return buffer;
}

char* ReadStringWithoutVowels(FILE* stream)
{
    if(feof(stream)) {
        return NULL;
    }

    const int chunkSize = 256;
    char* buffer = (char*)malloc(chunkSize);
    int bufferSize = chunkSize;

    if(buffer == NULL)
    {
        printf("Couldn't allocate buffer");
        exit(EXIT_FAILURE);
    }

    int readChar;
    int idx = 0;

    char* vowels = {"AEIOUYaeiouy"};

    while ((readChar = getc(stream)) != EOF)
    {
        int isVowel = 0;

        for (int i = 0; i < (int)strlen(vowels); ++i){
            if (readChar == vowels[i]){
                isVowel = 1;
```

```
            }
        }

        if (isVowel == 0){
            buffer[idx++] = readChar;
        }

        if (idx == bufferSize)
        {
            buffer = realloc(buffer, bufferSize + chunkSize);
            bufferSize += chunkSize;
        }

        if (readChar == '\n') {
            break;
        }
    }

    buffer[idx] = '\0';

    return buffer;
}
```

parent.c

```
#include "parent.h"
#include "utils.h"

int ChoosePipe(){
    return (int)rand() % 100 < 80;
}

void ParentRoutine(char* pathToChild, FILE* fin)
{
    char* fileName1 = ReadString(fin);
    char* fileName2 = ReadString(fin);

    fileName1[strlen(fileName1) - 1] = '\0';
    fileName2[strlen(fileName2) - 1] = '\0';

    unlink(fileName1);
    unlink(fileName2);
```

```
int fd1[2], fd2[2];

if (pipe(fd1) == -1 || pipe(fd2) == -1)
{
    perror("creating pipe error )");
    exit(EXIT_FAILURE);
}

pid_t outputFile1, outputFile2;

if ((outputFile1 = open(fileName1, O_WRONLY | O_CREAT, S_IRWXU)) < 0)
{
    perror("opening output file 1 error )");
    exit(EXIT_FAILURE);
}

if ((outputFile2 = open(fileName2, O_WRONLY | O_CREAT, S_IRWXU)) < 0)
{
    perror("opening output file 2 error )");
    exit(EXIT_FAILURE);
}

free(fileName1);
free(fileName2);

char* argv[2];
argv[0] = "child";
argv[1] = NULL;

pid_t pid1 = fork();
pid_t pid2 = 1;

if (pid1 > 0){
    pid2 = fork();
}

if (pid1 < 0 || pid2 < 0)
{
    perror("process error )");
    exit(EXIT_FAILURE);
}
```

```
if (pid1 == 0)
{
    close(fd1[1]);

    if (dup2(fd1[0], 0) < 0)
    {
        perror("duping pipe error )");
        exit(EXIT_FAILURE);
    }

    if (dup2(outputFile1, 1) < 0)
    {
        perror("duping output file error )");
        exit(EXIT_FAILURE);
    }

    execv(pathToChild, argv);

    //perror("execv error )");
    //exit(EXIT_FAILURE);
}
else if (pid2 == 0)
{
    close(fd2[1]);

    if (dup2(fd2[0], 0) < 0)
    {
        perror("duping pipe error )");
        exit(EXIT_FAILURE);
    }

    if (dup2(outputFile2, 1) < 0)
    {
        perror("duping output file error )");
        exit(EXIT_FAILURE);
    }

    execv(pathToChild, argv);

    //perror("execv error )");
    //exit(EXIT_FAILURE);
}
```

```
    else
    {
        close(fd1[0]);
        close(fd2[0]);

        char* strInput = NULL;

        while ((strInput = ReadString(fin)) != NULL)
        {
            int strSize = strlen(strInput);

            if (strSize > 0)
            {
                if (ChoosePipe())
                {
                    write(fd1[1], strInput, strSize);
                }
                else
                {
                    write(fd2[1], strInput, strSize);
                }
            }

            free(strInput);
        }

        if (strInput == NULL)
        {
            char terminator = '\0';

            write(fd1[1], &terminator, 1);
            write(fd2[1], &terminator, 1);
        }

        close(fd1[1]);
        close(fd2[1]);
    }
}
```

child.c

```
#include <stdio.h>
#include <stdlib.h>
```

```c
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <ctype.h>
#include <sys/wait.h>
#include <sys/stat.h>

#include "utils.h"

int main(int argc, char* argv[])
{
    if (argc < 1)
    {
        perror("too few arguments )");
        exit(EXIT_FAILURE);
    }

    if (strlen(argv[0]) < 1)
    {
        perror("too few arguments )");
        exit(EXIT_FAILURE);
    }

    char* strInput;

    while ((strInput = ReadStringWithoutVowels(stdin)) != NULL)
    {
        write(1, strInput, strlen(strInput));
        free(strInput);
    }

    return 0;
}
```

Выводы

Вызов *fork* дублирует породивший его процесс со всеми его переменными, файловыми дескрипторами, приоритетами процесса, рабочий и корневой каталоги, и сегментами выделенной памяти.

Ребёнок **не** наследует:

- идентификатора процесса (PID, PPID);
- израсходованного времени ЦП (оно обнуляется);

- сигналов процесса-родителя, требующих ответа;
- блокированных файлов (record locking).

В процессе выполнения лабораторной работы были приобретены навыки практического применения создания, обработки и отслеживания их состояния. Для выполнения данного варианта задания создание потоков как таковых не требуется, так как всю работу выполняет системный вызов «exec».